Abstract

We are interested in a natural generalization of term-rewriting techniques to what we call drags, viz. finite, directed, ordered, rooted multigraphs, each vertex of which is labeled by a function symbol. To this end, we develop a rich algebra of drags that generalizes the familiar term algebra and its associated rewriting capabilities. Viewing graphs as terms provides an initial building block for rewriting with such graphs, one that should impact the many areas where computations take place on graphs.

This paper is dedicated to the memory of Maurice Nivat, a colleague and friend of both authors. During his career, Maurice impacted many research areas, including formal language theory, combinatorics, semantics of programming languages, concurrency, and discrete geometry. His interest in formal languages lead him to study confluence properties of rewriting words. The present work follows along the path he cleared.

1. Introduction

Rewriting with graphs has a long history in computer science, graphs being used to represent data structures, but also program structures and even computational models. They play therefore a key rôle in program evaluation, transformation, and optimization, and more generally program analysis; see, for example, [7]. Since graphs originate from topological investigations, it is no wonder seeing them also play an important rôle in modern algebraic topology, more specifically in the theory of operads.

Our interest in graph rewriting originated from the needs of the study of operads, which has strong relationships with various areas of computer science, including concurrency theory and type theory. In the introduction to their book on the algebra of operads [10], Bremner and Dotsenko write:

Elements of operads are conventionally represented by linear combinations of trees, “tree polynomials”. Generalizations to algebraic structures where monomials are graphs that possibly have loops and are possibly disconnected, e.g. properads, PROPs, wheeled operads, etc., are still unknown, and it is not quite clear if it is at all possible to extend Gröbner-flavored methods to those structures.

We are accordingly interested in first-order terms with both sharing and back-arrows. These graphs, qua expressions, can be composed to form monomials. In turn, these monomials can be added to form polynomials, addition being associative and commutative. These finite graphs and their polynomials are seen as expressions of an algebraic structure that is referred to generically as an operad.

As rewriting graphs is very similar to rewriting algebraic terms, the same questions recur: What rewriting relation do we need? Is there an efficient pattern-matching algorithm? How can one determine if a particular rewriting system is confluent and/or terminating? And – last but not least – how should we represent graphs?
The above questions have, for these reasons, been addressed by the rewriting community since the mid-seventies in research initiated by Hartmut Ehrig and his collaborators [12]. Termination and confluence techniques have been elaborated for various generalization of trees, such as rational trees, directed acyclic graphs, lambda-terms, and lambda-graphs, as well as for graphs in general. But the design of a convenient mathematical framework for rewriting arbitrary graphs has made little progress beyond various categorical definitions and the study of the particular cases we just mentioned [27, 30]. See [17] for a survey of implementations of forms of graph rewriting and of available analysis tools.

This paper describes the design of a general class of graphs – actually, multigraphs – that has good structural properties with respect to rewriting. A specificity used for that purpose is the presence of arbitrarily many roots, as well as of variables labeling some leaves – called “sprouts”. Connecting the sprouts of one graph with the roots of another allows one to compose graphs in a particularly easy, natural way. Graphs in this class come equipped with a simple algebraic structure that allows one to view them as terms with sharing and cycles, and, therefore, to mimic many techniques that have been developed for trees over the years. In particular, rewrite rules become pairs of graphs with matching numbers of roots and with no variables on the right-hand side that do not also appear on the left-hand side. These constraints ensure that rewriting cannot create dangling pointers, as can be the case with the traditional approaches.

This work also opens the door for a notion of rewrite orderings for graphs and for syntactic termination proof methods that generalize the usual techniques used in term rewriting. Such generalizations were not previously possible in the presence of arbitrary cycles. These orderings are currently under exploration. See [9]. Confluence of these graph rewrite rules, on the other hand, has not been considered yet. The experience gained with the study of confluence for the categorical approach to graph rewriting should ease the way [24, 11, 13].

Despite being purely theoretical, the framework developed here should lead to easy implementations: the view of a drag as a term with sharing and cycles is extremely easy to implement, and suggests thereby a possibility to extend many existing term-rewriting implementations to actually do graph rewriting.

The particular class of graphs we are interested in is introduced in Section 2. The algebra of drags is described in Section 3, followed by the view of drags as dags in Section 4. Drag rewriting is introduced in Section 5 and explored further in Sections 6 and 7. Comparison with other frameworks comes next in Section 8. Section 9 presents our language of drag expressions. This is followed by a brief discussion.

2. Drags

The class of graphs with which we deal here is that consisting of finite directed graphs with labeled vertices, allowing multiple edges between vertices and an arbitrary number of roots. In the present work, we assume that the outgoing neighbors (vertices at the other end of outgoing edges) are ordered (from left to right, say, in the figures) and that their number is fixed, depending solely on the label of the vertex. Some vertices with no outgoing edges are designated sprouts. We shall call these finite directed rooted labeled graphs, drags.

We presuppose a set of function symbols Σ, whose elements \( f \in \Sigma \) will be used as labels for vertices, and which are each equipped with a fixed arity. (We have no associative-commutative symbols here.) In addition, we have a set of nullary variable symbols Ξ, disjoint from Σ, which will be used to label sprouts.

The successors of a vertex labeled \( f \) in a drag are implicitly interpreted as the arguments of the function symbol \( f \). The graph describes therefore the inverse of the computational flow. (Some might prefer the converse choice: this is a matter of taste and convention.) This class of graphs appears to be a very general model for describing computations in the absence of binding constructs.

In contrast with the usual categorical approaches to graph rewriting considered in Section 8, our multigraph model allows for a very standard conception of rewriting. The novelty, however, lies in that we allow for arbitrarily many roots and arbitrarily complex cycles: there is no restriction on the structure of the drags we deal with. This is because we view drags as networks of processing units (the vertices) that accept a given number of data as inputs and deliver some datum as output that can be sent over an arbitrary number of one-way channels. Channels can of course be duplicated, and there is an order among the input channels of a processing unit so as to appropriately discriminate among the inputs. Finally, the generality of the model requires that these one-way channels connect the processing units in an arbitrary way. All this makes our drag model very general.
To ameliorate notational burden, we will use vertical bars $|·|$ to denote various quantities, such as length of lists, size of sets or of expressions, and even the arity of function symbols. We use $\emptyset$ for an empty list, set, or multiset, $\cup$ set and multiset union, as well as for list concatenation, and $\setminus$ for set or multiset difference. We mix these, too, and denote by $K \setminus V$ the sublist of a list $K$ obtained by filtering out those elements belonging to a set $V$. We will also identify a singleton list, set, or multiset with its contents to avoid unnecessary clutter.

We were inconsistent in the order of the components of the tuple!

**Definition 1 (Draggs).** A drag is a tuple $⟨V, R, L, X, S⟩$, where

1. $V$ is a finite set of vertices;
2. $R : 1 .. |R| → V$ is a finite list of vertices, called roots, so $R(n)$ refers to the $n$th root in the list;
3. $S \subseteq V$ is a set of sprouts, leaving $V \setminus S$ to be the internal vertices;
4. $L : V → Σ$ is the labeling function, mapping internal vertices $V \setminus S$ to labels from the vocabulary $Σ$ and sprouts $S$ to labels from the vocabulary $Ξ$;
5. $X : V → V^*$ is the successor function, mapping each vertex $v ∈ V$ to a list of vertices in $V$ whose length equals the arity of its label (that is, $|X(v)| = |L(v)|$).

A drag is closed if it has no sprouts $S$, and open otherwise. The last component $S = \emptyset$ will often be omitted from the tuple for closed drags. Open drags act as patterns.

If $b ∈ X(a)$, then $(a, b)$ is a directed edge with source $a$ and target $b$. We also write $axb$. The reflexive-transitive closure $X^*$ of the relation $X$ is called accessibility. A vertex $v$ is said to be accessible from vertex $u$, and likewise that $u$ accesses $v$, if $uX^*v$. Vertex $v$ is accessible (without qualification) if it is accessible from some root. A root $r$ is maximal if any root that can access it is accessible from it. We denote by $R^*$ the set of maximal roots.

A drag is clean if all its vertices are accessible. An open drag is linear if no two sprouts have the same label. fix def of cyclic; removed: if no sprout is a root and And it is cyclic if every internal vertex can access some maximal root.

The labeling function extends to lists, sets, and multisets of vertices in the expected manner.

Terms as ordered trees, sequences of terms, and terms with shared subterms are all particular kinds of drags.

It will sometimes be convenient to consider roots as specific incoming edges and to identify a sprout with the variable symbol that is its label. We use these facilities unannounced.

The component $R$ of a drag is a list of roots possibly with repetitions, whereas $S$ is a set of sprouts. Having a list of roots, rather than a set, is one of the keys to a nice algebra of drags, and consequently for a general notion of rewriting. Sprouts may be roots, as we shall see in examples. This is important too for both generality and the algebraic structure. Nonlinear drags play an essential rôle in shared rewriting.

Given a drag $D = ⟨V, R, L, X, S⟩$, we make use of the following notations: $\mathit{Vert}(D) = V$ for its set of vertices; $X_D = X$ for its successor function; $\mathit{Acc}(D) = X^+(R)$ for its set of accessible vertices; $R(D) = R$ for its set of roots; $S(D) = S$ for its set of sprouts; $\mathit{Var}(D) = L(S)$ for the set of variables labeling its sprouts; $D^*_X$ to indicate its roots and sprouts; and $D^F$ for the clean drag obtained by removing inaccessible vertices and their related edges, which fits with the underlying intended behavioral semantics of drags. Note that cleanness, as can be expected, relates to garbage collection of unreachable vertices.

A cyclic drag does not necessarily have a single cycle going through all its vertices for at least three reasons: first, the union of cyclic drags is also cyclic; second, sprouts – if any – have no outgoing edges, so cannot participate in a cycle; third, an isolated root by itself is cyclic. A cyclic drag may, therefore, consist of a lone internal vertex that is a root. Although it may seem odd to refer to such an edgeless drag as cyclic, this is an essential ingredient here, one which helps us view a tree as a special case of a drag. For example, the ground (variable-free) term $a$ and the term $f(x)$ are both cyclic drags, whereas the terms $f(a)$ and $f(f(x))$ are noncyclic drags.

Ariola and Klop introduced a colorful terminology that we shall adopt when convenient: back arrows refer to edges pointing back up the path from the root, vertical sharing refers to edges ending up in some vertex further away from the root, and horizontal sharing is used for edges going sideways between paths.

A main concept used throughout is that of subdrag:

**Definition 2 (Subdrag).** Given a drag $D = ⟨V, R, L, X, S⟩$ and a subset $W ⊆ V$ of its vertices, its subdrag $D_W$ generated by $W$ is the drag $⟨V', R', L', X', S'⟩$, where

3
its vertices $V' \subseteq V$ are the least superset of $W$ that is closed under successors $X$;

its roots $R'$ are $(R \cap V') \cup (X(V \setminus V') \cap V')$; and

$L', X', S'$ are the restrictions of $L, X, S$ to that $V'$.

A subdrag is clean by construction. Its roots include as new roots those vertices in $V'$ that had an incoming edge in $D$ that is no longer in the subdrag $D_{|W}$, with the corresponding multiplicity. The order of those additional roots in $R'$ does not really matter for now.

In particular, restricting a drag $D$ to its maximal roots, yields $D^\sharp$. On the other hand, restricting any drag to an empty set of vertices, yields the empty drag $\emptyset \equiv 1_\emptyset = (\emptyset, \emptyset, \emptyset, \emptyset)$.

Drags are graphs. An isomorphism between two open drags is a one-to-one mapping between their respective sets of (accessible) vertices that identifies their respective labels (up to renaming of the sprouts’ labels done here by the very same mapping) and lists of roots, and commutes with their respective successor functions.

**Definition 3** (Drag Isomorphism). Given two open drags $D = \langle V, R, L, X, S \rangle$ and $D' = \langle V', R', L', X', S' \rangle$ whose sprouts are identified with their labels, an isomorphism from $D$ to $D'$ is a one-to-one mapping $o : \text{Acc}(V) \leftrightarrow \text{Acc}(V')$ such that

1. mapping $o$ restricts to bijections between the lists of roots and the sets of accessible sprouts;
2. mapping $o$ respects labels and successors of internal vertices: $\forall v \in \text{Acc}(V) \setminus S$. $L(v) = L'(o(v))$ and $X(v) = X'(o(v))$.

We write $D \equiv_o D'$.

It is sometimes useful to consider multisets instead of lists of roots, resulting in a coarser equivalence on drags. Accordingly, we write $D \equiv_o D'$ and say that $D, D'$ are quasi-isomorphic if $o$ restricts to a bijection between the multisets (instead of lists) of roots.

We write $D \equiv D'$ and $D \equiv D'$ instead of $D \equiv_o D'$ and $D \equiv_o D'$ in case $o$ is the identity.

Identifying sprouts with the variables that label them saved an additional one-to-one mapping between variables that would otherwise have been necessary.

Note that inaccessible sprouts are ignored in the definition of drag isomorphism, making any drag $D$ isomorphic to its clean version $D^\sharp$.

**Lemma 4.** For any drag $D$, $D \equiv D^\sharp$.

**Example 5.** Consider the seven single-rooted drags depicted in Figure 1 with the convention that the order of arguments of function symbols begins with the leftmost outgoing edge and continues counterclockwise. An incoming down-arrow indicates a root.

Let $\Sigma = \{a, b, f, g, h\}$ with $|f| = 3$, $|b| = 2$, $|g| = 1$, $|a| = |b| = 0$. We first consider the term $f(g(b), a, g(b))$ in which $b$ or $g(b)$ can be shared. Among all possible cases, three are represented. The fifth has both horizontal and vertical sharing. The sixth has three cycles: an independent leftmost loop and a horizontal cycle sharing another horizontal cycle below. The last has a single big cycle, which is both horizontal and vertical including an inner cycle.
3. Drag algebra

We equip drags with a composition operator, which combines two drags by connecting the sprouts of each to the roots of the other. This will be a central notion for us, one which makes drag model compositional. The idea is quite intuitive: take the uses of the links, and connect the sprouts and roots according to a device we call a switchboard.

As usual, we denote by $\text{Dom}(\xi)$ and $\text{Im}(\xi)$ the domain (of definition) and image of a (partial) function $\xi$, using $\xi_{A\rightarrow B}$ for its restriction going from subset $A$ of its domain to subset $B$ of its image, omitting $\rightarrow B$ when irrelevant. Both $\text{Dom}(\xi)$ and $\text{Im}(\xi)$ are sets, hence $\text{Im}(\xi)$ may differ from $\xi(R)$, which is a list, possibly with repetitions when $R$ is a list itself.

We begin with the connection device:

**Definition 6 (Switchboard).** Let $D = \langle V, R, L, X, S \rangle$ and $D' = \langle V', R', L', X', S' \rangle$ be open drags. A switchboard $\xi$ for $D, D'$ is a pair $\langle \xi_D : S \rightarrow 1..|R|, \xi_{D'} : S' \rightarrow 1..|R| \rangle$ of partial injective functions (we also say that $D' \xi$ is an extension of $D$), such that

1. $s \in \text{Dom}(\xi_D)$ and $L(s) = L(t)$ imply $t \in \text{Dom}(\xi_D)$ and $D'_{\text{Dom}(\xi_D)(s)} \simeq D'_{\text{Dom}(\xi_D)(t)}$ for all sprouts $s, t \in S$;
2. $s \in \text{Dom}(\xi_{D'})$ and $L'(s) = L'(t)$ imply $t \in \text{Dom}(\xi_{D'})$ and $D_{\text{Dom}(\xi_{D'})(s)} \simeq D_{\text{Dom}(\xi_{D'})(t)}$ for all sprouts $s, t \in S$.

A switchboard $\xi$ is well-behaved if it does not induce any cycle among sprouts, using $\xi, R, R'$ relationally:

$$\exists n > 0, s_1, \ldots, s_{n+1} \in S, t_1, \ldots, t_n \in S', s_1 = s_{n+1}, \forall i \in 1..n, s_i \xi D'R^X t_i \xi D'RX^* s_{i+1}$$

Both conditions (1,2) are always satisfied for linear drags, or for nonlinear drags whose switchboard, called linear, is defined for sprouts whose variables are all different. When this is not the case, checking that a particular $\xi$ is a switchboard involves graph isomorphism, which is polynomial in our case, since we have ordained fixed arities. (Were one to allow variadic vertices, isomorphism would be quasi-polynomial [3].) A simple effective approximation here is to require instead that $R'(\xi_D(s)) = R'(\xi_D(t))$ for [1] and similarly $R(\xi_{D'}(s)) = R(\xi_{D'}(t))$ for [2]. This approximation fits well with shared trees, for which isomorphism is computable in linear time.

In the sequel, we assume this latter condition for simplicity and will discuss its impact when appropriate, in particular in the conclusion.

In practice, we shall usually enforce $\text{Var}(D) \cap \text{Var}(D') = \emptyset$ so as to avoid confusion, but this is by no means necessary since switchboards are pairs of mappings operating on $\text{Var}(D)$ and $\text{Var}(D')$ independently, not a single mapping operating on $\text{Var}(D) \cup \text{Var}(D')$. Further, splitting $\xi$ makes the definition clearly symmetric.

Note finally that injectivity of $\xi$ and the fact that roots are lists with repetitions get along together. Injectivity implies that the list $\xi_{D'}(\text{Dom}(\xi_D))$ is indeed a set, hence making the set difference $[1..|R'|] \setminus \xi_{D'}(\text{Dom}(\xi_D))$ well defined.

A switchboard induces a binary operation on open drags. The essence of this definition is that the (disjoint) union of the two drags is formed, but with sprouts in the domain of the switchboards merged with the roots to which the switchboard images refer. This operation removes one sprout and one root at the same time from the sets of sprouts and lists of roots of the union. The number of roots in the union is obtained, therefore, by summing the number of roots of the two drags and then subtracting the number of roots removed by the switchboard, that is, those in its domain. Of course, one needs to worry about the case where multiple sprouts are merged successively, when the switchboards map sprout to rooted-sprout to rooted-sprout, possibly in a cycle. In such cases, all need to be renamed to the same one, while accumulating root statuses. Computing that chosen one is the rôle of the coming recursive definition in which an additional argument is used to collect all sprout names along a path back and forth in the switchboard, so as to check for the existence of a cycle among them:

**Definition 7 (Target).** Let $D = \langle V, R, L, X, S \rangle$ and $D' = \langle V', R', L', X', S' \rangle$ be drags such that $V \cap V' = \emptyset$, and $\xi$ be a switchboard for $D, D'$. The target $\xi'(s)$ of a sprout $s \in S \cup S'$ is a vertex $v$ given by target$(s, \{s\})$, where target$(s, T)$ is defined as follows:

Let $v = R'(n)$ if $s \in S$, and $v = R(n)$ if $s \in S'$, where $n = \xi(s)$.

1. If $v \in (V \cup V') \setminus (S \cup S')$, then target$(s, T) = v$.
2. If $v \in (S \cup S') \setminus \text{Dom}(\xi)$, then target$(s, T) = v$. 

5
3. If \( v \in \text{Dom}(\xi) \setminus T \), then target\((s, T) = \text{target}(v, T \cup \{s\}). \)

4. If \( v \in T \), then target\((s, T) = t \), where \( t \in T \) is canonically determined.

The target \( \xi(\cdot) \) is extended to all vertices by letting \( \xi^*(v) = v \) when \( v \in (V \setminus S) \cup (V' \setminus S') \). We denote by \( \xi \downarrow \) the subset of \( S \cup S' \) whose sprouts are also targets (given by cases 2 and 4 above) and by \( |\xi|_S \) the number of cycles among sprouts induced by \( \xi \) (given by case 4 above).

One may wonder if target\((s, \{s\}) \) computes (in its second argument) the whole set \( T \) of sprouts that can be reached from the sprout \( s \) in case \( \xi \) induces a cycle among the sprouts. This is so because \( \xi \) is injective, which implies that these cycles must be by elementary paths. Of course, \( T \) may contain sprouts that are not part of the cycle, from which the cycle can be accessed. The choice of a target sprout among those that belong to the cycle \( C \) contained in \( T \) is then arbitrary, provided it is a function taking as input a nonempty subset \( T \) of \( S \cup S' \) and returning as output an element of \( S \cap C \). The cycle contained in \( T \) must contain sprouts from both drags \( S', S'' \). (We could have chosen an element of \( S' \cap C \) instead, but prefer to privilege drag \( D \) over drag \( D' \).) All these technicalities become much simpler if the switchboard \( \xi \) is well-behaved, in which Case 2 becomes void.

Notice that \( \xi^*(s) \) is a vertex name (possibly a sprout name) and that \( \xi \downarrow \) is a set of vertex names that are all sprout names. They are not positions in the lists of roots of \( D, D' \), as is the switchboard \( \xi \). This choice aims at easing the calculation of the list of vertices in the definition of composition.

We are now ready for defining the composition of two drags. Its set of vertices will be the union of two components: the internal vertices of the drags, and their sprouts which are target vertices. Sprouts and labeling will be given by restricting the original ones to the set of vertices that is obtained.

A difficulty pops up with the list of roots of the composed drag. There are two problems here: first, each root number of \( D \) or \( D' \) belonging to the image of \( \xi \) will disappear, creating holes in the corresponding list of roots, which will need to be eliminated from the composition; second, roots numbers from \( D' \) must be translated so as to become root numbers of the composed drag, assuming the roots from \( D' \) come second, which is our choice. The correspondence between these numberings will be given by the function \( \eta \), so that, if \( n \) is a root number in the composition, \( \eta(n) \) is the corresponding root number in \( D \) or \( D' \), depending on the value of \( n \). The list of roots that is obtained is accordingly the concatenation of both original lists of roots deprived beforehand from the roots in the image of the switchboard, except that each sprout from the domain of \( \xi \) that remains in that list must be replaced by its target. Having extended the target function to all vertices makes it easier.

**Definition 8 (Composition).** Let \( D = (V, R, L, X, S) \) and \( D' = (V', R', L', X', S') \) be drags such that \( V \cap V' = \emptyset \), and \( \xi \) be a switchboard for \( D, D' \). Their composition is the drag \( D \otimes \xi D' \equiv (V'', R'', L'', X'', S'') \), where

1. \( V'' = (V \setminus S) \cup (V' \setminus S') \cup \xi \downarrow \).
2. \( R'' : \{1..|R| + |R'| - |\text{Dom}(\xi)| \} \rightarrow V'' \) such that \( R''(n) = \xi^*(R(n + \eta(n))) \) if \( n \in 1..(|R| - |\text{Dom}(\xi)|) \); otherwise \( \xi^*(R(n - (\text{Dom}(\xi))) + \eta(n + (\text{Dom}(\xi)))) \), where the relocation function \( \eta \) counts the number of roots used up so far by \( \xi \) in the corresponding list of roots: \( \eta(0) = 0 \) and \( \eta(n) = \xi_0(n, \eta_0(n - 1)) \), with \( \xi_0(n, p) = p \) if \( R(D(n + p)) \notin \text{Im}(\xi) \); otherwise \( \xi_0(n, p + 1) \).
3. \( S'' = (S \setminus \text{Dom}(\xi)) \cup (S' \setminus \text{Dom}(\xi')) \cup \xi \downarrow \).
4. \( L''(\xi(v) \in V \setminus V') = L(v) \) and \( L''(v \in V' \setminus V'') = L'(v) \).
5. \( X''(\xi(v) \in V' \setminus S) = \xi(X(v)) \) and \( X'''(\xi(v) \in V' \setminus S') = \xi'(X'(v)) \).

It is easy to see that \( D \otimes \xi D' \) is a well-defined drag, that is, it satisfies the arity constraint required at each vertex.

Note also that, if \( \xi \) is surjective and \( \xi'' \) total, a category of switchboard that will play a key rôle for rewriting, then all roots and sprouts of \( D' \) disappear in the composed drag. (If \( \xi'' \) is surjective and \( \xi'' \) total, those from \( D \) disappear.) Otherwise, the symmetry of the definition is broken by choosing the roots originating from \( D \) to come first.

**Example 9.** We show in Figure 2 three examples of compositions, the first two with similar drags. The first composition is a substitution of terms. The second uses a bi-directional switchboard, which induces a cycle. In the second example, the remaining root is the first (red) root of the first drag which has two roots, the first red, the other black. The third example shows how sprouts that are also roots connect to roots in the composition (colors indicate roots’ origin). Note that the second root of the sprout labeled by \( y \) (number 2 of the drag) has disappeared in the composition, while its first root (number 3 of the drag) is now the fourth result. This agrees with the definition, as shown by...
the following calculations (naming the vertices by their label without ambiguity in this example): \( \xi^*(x) = \xi^*(y) = x; \) 
\( \xi^* \downarrow (x); \) \( \text{Dom}(R) = 1..3, \text{Dom}(R') : [1,3], \text{Dom}(R'') = 1..4 (4 = 3 + 3 - 2); \) \( \eta(1) = 1, \eta(2) = 2, \eta(3) = 1, \eta(4) = 2. \) 
Note that the latter switchboard is not well-behaved, since the target calculation results in a sprout.

It is easy to characterize the set of variables of a drag obtained by composition in the case the switchboard is well-behaved:

**Lemma 10.** Let \( A, B \) be two drags whose multisets of variables are disjoint, and \( \xi \) a well-behaved switchboard. Then, we have the following equalities between multisets of variables:

\[ \{ \text{Var}(A \otimes \xi B) = (\text{Var}(A) \cup \text{Var}(B)) \setminus (\text{Dom}(\xi) \cup \text{Im}(\xi)) \} \]

and among lists of roots:

\[ R'' : [1..|R| + |R'| - |\text{Dom}(\xi)|] \rightarrow V'' \]

such that

\[ R'' = R([1..|R| \setminus \xi \text{Pr}(\text{Dom}(\xi))) \cup R'([1..|R'| \setminus \xi \text{Pr}(\text{Dom}(\xi))) \]

**Proof.** This is so because of the absence of cycles among the variables. \( \Box \)

Lemma [10] shows that the calculation of the list of roots of a composition of drags becomes quite simple when the switchboard is well-behaved. As we shall see later, switchboards must be well-behaved when it comes to drag rewriting. However, we do not want to commit ourselves to well-behaved switchboards at this point.

Our definition of switchboard being (almost) symmetric, composition is itself symmetric provided all roots of the resulting drag originate from the same side. Otherwise, composition yields drags that are equal up to cyclic permutation of their roots.

**Lemma 11 (Commutativity).** Composition of drags is quasi-commutative, and becomes commutative for switchboards whose context or substitution is surjective, such as rewriting switchboards.

Commutativity shows that there is no big difference between the context and the substitution of a switchboard in a composition \( D \otimes_{\xi} D' \), since they simply exchange each other when exchanging the input drags of the composition.

A linear open drag all of whose vertices are its sprouts, whose set of edges is empty, and whose list of roots is a list made of its sprouts is called an identity. We denote it \( 1^X_{\xi} \), where \( X \) is its set of sprouts and \( Y \) is a list of elements of \( X \). An identity drag is therefore clean if all its sprouts occur in its list of roots. We use \( \emptyset \) for the clean drag \( 1^\emptyset_{\emptyset} \), called the empty drag.

Given a drag \( D \), an identity extension of \( D \) is the pair \( 1^X_{\xi} \), made of an identity drag \( 1^X_{\xi} \), such that \( X \subseteq \text{Var}(D) \), and an identity switchboard \( \xi \), such that \( \text{Dom}(\xi) = X, \iota_D \) is the identity (abusing our notations) and \( \text{Dom}(\iota_{1^X_{\xi}}) = \emptyset. \)

**Lemma 12 (Neutrality).** Let \( D \) be an open drag, \( X \subseteq \text{Var}(D) \), and \( 1^X_{\xi} \) an identity extension for \( D \). Then \( 1^X_{\xi} \otimes_D \emptyset \emptyset = D \), while \( D \otimes_{\emptyset} 1^X_{\xi} = D \).

In particular, this property holds when \( X = \emptyset \), in which case \( D \otimes_{\emptyset} 1^\emptyset_{\emptyset} = D \). In case \( X \neq \emptyset \), then the sprouts of the composition \( X \neq \emptyset \) will be those of the identity drag, not those of \( D \).

**Definition 13 (Compatibility).** Two switchboards \( \xi \) and \( \zeta \) for the respective pairs of drags \( A, B \) and \( B, C \) are compatible if \( \text{Dom}(\xi_B) \cap \text{Dom}(\xi_B) = \emptyset \) and \( \text{Im}(\xi_A) \cap \text{Im}(\xi_C) = \emptyset. \)
Lemma 14 (Associativity). Let $A, B, C$ be three drags, and $\xi, \zeta$ be compatible switchboards for $A, B$ and $B, C$, respectively. Then, $(A \otimes \xi B) \otimes \zeta C = A \oplus \xi (B \otimes \xi C).

Proof. Compatibility ensures that $\xi$ and $\zeta$ are switchboards for $A, B \otimes \xi C$ and $A \otimes \xi B, C$, by eliminating the possibility that a sprout or root is used twice. Both sides of the associativity equation are thus defined. The equality itself results from easy checking. $

The assumption that $\xi$ and $\zeta$ are compatible switchboards for $A, B$ and $B, C$ is important here. Associativity does not apply to any expression of the form $(A \otimes \xi B) \otimes \zeta C$, in which $\otimes \xi$ has two arguments $A \otimes \xi B$ and $C$, instead of $B$ and $C$ as assumed in the statement.

4. Drag structure

Next, we investigate ways of decomposing a drag, so as to exhibit its tree-like structure. This question should have many applications that remain to be explored. One of the applications should be the design of efficient algorithms for graph matching.

We first give a general construction that splits a drag into the subdrag generated by some of its vertices, and the rest of the drag, its antecedent. This requires defining a specific kind of extensions that generalizes the one we have defined for identity drags:

Definition 15 (Directed Switchboard). A switchboard $\xi$ for $D$, $D'$ is directed if one of $\xi_D$ and $\xi_{D'}$ has an empty domain. The pair $D' \xi_{D'}$ is a context extension of $D$ if $\text{Dom}(\xi_D) = \emptyset$ and a substitution extension of $D$ if $\text{Dom}(\xi_{D'}) = \emptyset$.

Directed switchboards correspond to the tree case, with all connections going from one drag to the other. Note that context extensions and substitution extensions switch with each other by symmetry of the definition of a switchboard.

Lemma 16. Given a drag $D$ and a subset $W \subseteq V$ of its vertices, there is a drag $A$, called the antecedent, and a directed linear switchboard $\zeta$, such that $D = A \otimes \zeta D_W$.

Proof. For internal vertices, $A$ has those vertices of $D$ that do not belong to $D_W$. Its sprouts are the sprouts of $D$ that are not in $D_W$, plus new sprouts that correspond to the new roots of $D_W$, that is, to the vertices of $D_W$ whose antecedents in $D$ are not vertices of $D_W$. These sprouts can be labeled by different variables so as to make $A$ linear on them, hence ensuring unicity of the antecedent up to isomorphism. (We could also ensure unicity by labeling any two of these sprouts by the same variable iff they generate equal drags – or, perhaps, isomorphic drags – in $D$.) The roots of $A$ are those of $D$ that are not vertices of $D_W$, plus the new sprouts that become roots with the appropriate order of multiplicity (the corresponding number of outgoing edges to $D_W$ in $D$). The switchboard $\xi$ mapping these sprouts to the corresponding roots of $D_W$ is linear as a consequence of the labeling, directed since subdrags are closed under successor, and total on the new roots. Note that the rooted-sprouts of $A$ disappear in the composition.

The fact that the switchboard $\xi$ is directed expresses the property that decomposing a drag into a subdrag and its antecedent does not break any of its cycles – by the definition of subdrag. Note also that $\xi$ induces an order on the roots of the subdrag that are not roots of the whole drag. If $W = V$, then $A \xi$ must be the identity extension $1_x^V$ with $|X| = |R(D)|$.

A tree headed by the symbol $f$ of arity $n$ has one root labeled $f$, or equivalently a head corresponding to the expression $f(x_1, \ldots , x_n)$, plus several subtrees, seen here as a single drag with many roots. This unique decomposition is a fundamental property of trees that expresses the fact that the set of trees equipped with “head-operations” has an initial algebra structure.

Likewise, a drag has a head, which is intended to be the largest cycle containing the maximal roots of the drag – also the smallest nontrivial antecedent of the drag – and one tail, which is a list of several connected components. The head cycle may consist of a lone (root) vertex, in particular in the case of trees. As for trees, the head will have a list of new sprouts that are in one-to-one correspondence with the roots of the subdrag. A drag can therefore be seen as a bipartite graph made of its head, its tail, and a directed switchboard specifying that correspondence.

Definition 17 (Tail). The tail $\nabla D$ of a drag $D = \langle V, R, L, X, S \rangle$ is the subdrag generated by the set of vertices $V \setminus \{ v \in V : vX^+R^*(D) \}$. 

8
As an application of Lemma 16 we get:

Lemma 18 (Head). Given a drag $D$, there exists a drag $\hat{D}$, called head of $D$, and a directed linear switchboard $\xi$, such that $D = D \otimes_\xi \nabla D$.

The head of a drag is therefore the antecedent of its tail. By Lemma 16 it contains all maximal roots of $D$. Note further that its sprouts that were not already sprouts of $D$ are linear and that $\xi$ is total on those sprouts. More precisely, $\xi$ is a bijection between those sprouts and the roots of the tail that were not already roots of the drag $D$. In the sequel, we assume that the sprouts of the head are ordered with respect to a depth first search initialized with its list of roots.

This order on the sprouts of the head induces via the switchboard $\xi$, an order on the roots of the associated tail that were not roots of the original drag. The tail is therefore now canonically determined.

We could have defined the head and tail of $D$ as a pair of drags whose head contains all maximal roots, is cyclic, and its composition with the tail by a directed linear switchboard is the drag $D$ itself. This characterization of the head of a drag is a property that we should keep in mind.

We can now show the structure theorem of open drags:

Theorem 19 (Structure). Isomorphic drags have isomorphic heads and tails.

Proof. By Lemma 4 we restrict our attention to clean drags. Let $H \otimes_\xi T \defeq \langle V, R, L, X, S \rangle = o \langle V', R', L', X', S' \rangle \defeq H' \otimes_\xi T'$, where $H$ and $H'$ are heads, $T$ and $T'$ are open drags, $\xi_I$ and $\xi_{I'}$ are empty, $\xi_H$ and $\xi_{H'}$ are total and surjective, and $o$ is a bijection (identifying sprouts with their labels). We build new bijections $o_h$ and $o_t$ that witness the isomorphisms between heads and tails, respectively.

Since $\xi_I$ is empty and $\xi_H$ is surjective, the roots of $H \otimes_\xi T$ are those of $H$. Likewise, the roots of $H' \otimes_\xi T'$ are those of $H'$. Hence $o$ restricts to a one-to-one mapping from the roots of $H$ to those of $H'$. By property 3 of isomorphisms, $o$ restricts to a one-to-one mapping from the vertices of $H \otimes_\xi T$ that are vertices of $H$ to the vertices of $H' \otimes_\xi T'$ that are vertices of $H'$, and satisfies 3. To extend $o_h$ to the sprouts of $H$ and $H'$, it suffices to notice that they are equally many by 2. Since $o$ satisfies 3, so does $o_h$, hence $H = o_h H'$.

We can now easily define $o_t$ by removing $o_h$ from $o$. The roots of $T$ and $T'$ correspond to the sprouts of $H, H'$. Since the roots of $T$ and $T'$ are in one-to-one correspondence with the sprouts of $H$ and $H'$, respectively, $o_t$ restricts to the roots of $T, T'$. Also, since $\xi_H$ and $\xi_{H'}$ are surjective, the sprouts of $T$ and $T'$ are those of $H \otimes_\xi T$ and $H' \otimes_\xi T'$, respectively. Hence, $o_t$ restricts to the sprouts of $T, T'$. Since $o$ satisfies 3, so does $o_t$, hence $T = o_t T'$.

Example 20. The successive tails of the drags of Figure 1 are displayed in Figure 3. The numbering displayed above the roots of each tail specifies the order of roots in their list of roots. These tails, represented on the first row, have all the same number of roots, which is determined by their head $f$. The next row represents the next level of tails, which here have one or two roots, depending on their own head. The last tail in that row has two roots, a nonmaximal one in its father drag, and a new one.

The fundamental property of a drag expressed by Theorem 19 is that its decomposition into head and tail faithfully represents it. This property holds true because drags are multirooted. Uni-rooted drags cannot represent horizontal
sharing, in contrast to vertical sharing, which can sometimes be preserved with uni-rooted drags. Moreover, the fact that tails are quasi-isomorphic, rather than isomorphic, does not hamper faithfulness: different orders of new roots for the different possible tails yield different switchboards, but the order of roots resulting from the composition does not depend upon those orders since the new roots disappear in the composition.

Given a drag $D$, we can therefore partition its set of vertices into those belonging to a connected component of the vertices belonging to its head, and, recursively, the partition associated with its tail. The successor function of $D$ can then be seen as a successor function between the elements of the partition, hence defining a multigraph which is acyclic, hence is a dag (actually, a multi-rooted multi-dag) which is a dag decomposition of $D$ \cite{Section21}. This dag decomposition, however, does not have a bounded width in general, hence does not provide, in our case, polynomial time algorithms for queries expressible in second-order monadic logic. It will nevertheless play an important rôle in Section\textsuperscript{21}.

Let $C$ be all maximal strongly connected components of drag $D$. The dag decomposition of $D$ has vertices $C$ and as edges those that cross components. The head $H(D)$ of the drag comprises the components containing the maximal roots. The tail $T(D)$ is the rest. The decomposition is $(H(T'(D)))$.

5. Drag rewriting

Rewriting with drags is very similar to rewriting with trees: we first select an instance of the left-hand side $L$ of a rule in a drag $D$, then replace it by the corresponding right-hand side $R$. Selecting a left-hand side $L$ in a drag $D$ amounts to expressing the drag $D$ as a composition of some drag $W$ with $L$ via a switchboard $\xi$. Replacement amounts to computing the composition of $W$ with the right-hand side $R$ via the same switchboard $\xi$. A very important condition for the result to be a drag is, accordingly, that left-hand and right-hand sides of rules have the same number of roots. Our definition will therefore avoid the creation of dangling pointers.

Definition 21 (Rules). A graph rewrite rule is a pair of open clean drags written $L \rightarrow R$ such that $|\mathcal{R}(L)| = |\mathcal{R}(R)|$ and $\text{Var}(R) \subseteq \text{Var}(L)$. A graph rewrite system is a set of graph rewrite rules.

Trees are particular clean uni-rooted drags. Because they have a single root, term-rewrite rules satisfy the first condition. The second must be explicitly stated in the definition of a term-rewriting rule, as for drags.

Note that we could allow $R$ to be unclean, and strengthen instead the second condition to be $\text{Var}(R) = \text{Var}(L)$. Adding unreachable sprouts to a clean $R$ would then do with an isomorphic right-hand side. Such a handy trick – which we do not use here – is not possible with trees. In the conclusion, we discuss another application of using drag isomorphism for defining the result of a rewrite step.

Rewriting drags uses a specific kind of switchboard, which allows one to “encompass” a drag $U$ within drag $D$, so that all roots and sprouts of $U$ disappear from the composition:

Definition 22 (Rewriting Switchboard). A well-behaved switchboard $\xi$ for $W, U$ is a rewriting switchboard if $\xi_w$ is linear and surjective and $\xi_U$ is total. The pair $W, \xi$ is called a rewriting extension of $U$.

When $D = W \otimes_U U$, we also say that $D$ matches $U$, $W$ and $\xi$ being the matching context and switchboard. Matching is therefore the operation which, given $D$ and $U$, computes the drag $W$ and switchboard $\xi$ such that $D = W \otimes_U U$.

We are now ready for defining rewriting:

Definition 23 (Rewriting). Let $\mathcal{R}$ be a graph rewrite system. We say that a nonempty clean drag $D$ rewrites to a clean drag $D'$, and write $D \rightarrow_{\mathcal{R}} D'$ iff $D = W \otimes_{\xi} L$ and $D' = (W \otimes_{\xi} R)'$ for some drag rewrite rule $L \rightarrow R \in \mathcal{R}$ and rewriting extension $W \xi$ of $L$ such that $\xi_w$ is linear.

All assumptions on $\xi$ play a rôle.

First, the switchboard must be well-behaved. Were it not, then Example\textsuperscript{24} below shows that rewriting may generate fresh variables. We shall see that this is not possible with well-behaved switchboards. Second, because $\xi$ is a rewriting switchboard, $\xi_w$ must be linear. This implies that the variables labeling the sprouts of $W$ that are not already sprouts of $D$ must be all different. Third, $\xi_U$ must be surjective, implying that the roots of $L$ disappear in the composition. Fourth, $\xi_L$ must be total, implying that the sprouts of $L$ disappear in the composition. Fourth, $D$ must be nonempty, implying that the roots of $W$ do not all disappear in the composition; hence $\xi_L$ could not be surjective.
Note further that, if $L$ is linear, then $ξ$ is a linear rewriting switchboard: no test for equality (or, more generally, isomorphism) is needed when pattern matching $D$ with $L$ in this case.

If left-hand and right-hand sides of a drag rewrite rule have the same variables as sprouts, the computed drag generated will be clean if the starting drag is. But if there are strictly fewer variables in the right-hand side, replacement isomorphism) is needed when pattern matching $D$ respectively. The drag $D = \langle \{ x' \mapsto 1, y' \mapsto 2 \} \rangle \& f$.

Example 24. Consider the cyclic composition of Figure 2, and let $f(y') \rightarrow y'$ be the graph rewrite rule whose left-hand and right-hand sides are the drags $\langle f_1, f_2, f_3, f_4 \mapsto \rangle$, respectively. The drag $D = \langle f_1, f_2, f_3, f_4 \mapsto \rangle$ rewrites to the drag $D' = \langle f_1, f_2, f_3, f_4 \mapsto \rangle$.

It is also possible to break a cycle in a drag: the same drag $D$ rewrites to the drag $D'' = \langle f_1, f_2, f_3, f_4 \mapsto \rangle$.

These two rewrites are shown in Figure 2. In both cases, the upper occurrence of $f$ (in blue) is part of the context, whereas the one below (in red) is part of the rewrite rule. Determining what happens when rewriting the upper $f$ instead of the lower is left to the interested reader.

Instead, we rewrite the drag made of a single node (its root) labeled $f$, whose successor is itself, and consider also ill-behaved switchboards. We get: $\langle f_1, f_2, f_3, f_4 \mapsto \rangle = f(y') \ast (y') \ast \langle y_1, y_1, y_1 \mapsto \rangle \ast \langle y_1, y_1, y_1 \mapsto \rangle$.

This last (ill-behaved) rewrite is pictured in the third row of Figure 2.

Lemma 25. If $U \rightarrow_R U'$, then $\text{Var}(U') \subseteq \text{Var}(U)$ and $|\text{R}(U)| = |\text{R}(U')|$.

Proof. Both properties are true of rewrite rules. The second is preserved by rewriting with arbitrary switchboards. The first is not preserved by rewriting with ill-behaved switchboards, as we have seen. But it is preserved by rewriting with well-behaved switchboards, hence with rewriting switchboards, as a consequence of Lemma 10.

Lemma 26. Assume $U, U', W$ are three drags such that $U$ rewrites to $U'$, and $ξ = (ξ_W, ξ_U)$ is a switchboard for $W, U$. Then, $ξ' = (ξ_W, ξ_{U'})$ – also denoted $ξ$, where $ξ_U$ is the restriction of $ξ_U$ to $\text{Var}(U')$, is a switchboard for $W, U'$.
now merged into a single sprout labeled \( z \in V \) a clean nonempty drag, and for all \( t \)

Definition 27

(Cyclic Extension)

sharing among its sprouts. Identity sharing extensions are complete, in the sense that any sharing that is introduced

\[ U \]

is an identity drag and \( \xi \) of \( U \) \( D \)

therefore (approximately) a substitution extension of \( U \); those that are not accessible from the sprouts of \( U \) form the remaining part \( A \), which is therefore (approximately) a context extension of \( U \).

Particular substitution extensions of \( U \) are identity sharing extensions \( 1^{\xi}_U \) such that \( \xi(x_1) = \cdots = \xi(x_n) = z \), where \( x_1, \ldots, x_n \) are variables labeling sprouts of \( U \), and, of course, \( z \notin \text{Dom}(\xi) \). Then, the drag resulting from the composition of \( U \) with that extension is the same as \( U \), except that all its sprouts labeled by the variables \( x_1, \ldots, x_n \) are now merged into a single sprout labeled \( z \). The rôle of these extensions is to modify the structure of \( U \) by introducing sharing among its sprouts. Identity sharing extensions are complete, in the sense that any sharing that is introduced by a substitution extension can be achieved by an identity sharing extension.

We are left with the task of defining the kind of extension that \( B \) belongs to:

Definition 27 (Cyclic Extension). An extension \( B \xi \) of a clean drag \( U \) is cyclic if \( B \) is generated by \( \text{Im}(\xi_U) \), \( B \Theta_\xi U \) is a clean nonempty drag, and for all \( t \in \text{Ver}(B) \), there exists \( s \in \text{Dom}(\xi_B) \) such that \( tX_Bs \). The extension is trivial if \( B \) is an identity drag and \( \xi_B = \emptyset \), and total if \( \xi_B \) is surjective (\( B \) being otherwise arbitrary).

This condition resembles the one for cyclic drags, but does not always imply that \( B \Theta_\xi U \) is cyclic because not all internal vertices of \( U \) may reach a root of \( B \Theta_\xi U \). Only total extensions are cyclic, up to possibly ground subdrags of \( U \) when there are such subdrags in \( U \).

The conditions for being a cyclic extension impose that \( \xi_U \) is onto \( \mathcal{R}(B) \) so as to generate \( B \). The roots of the resulting drag are therefore originating from either \( B \) or \( U \) (possibly via a transfer), but there must be sufficiently many of them so that the resulting drag is clean.

Identity cyclic extensions of \( U \) are of the form \( 1^Z_U t \), where the variables in \( Z \subseteq Y \) are one-to-one with those in \( \text{Dom}(t_U) \subseteq \text{Var}(U) \), and \( t_U : Y \to 1 \ldots |R| \) is an arbitrary map. The rôle of cyclic extensions is to modify the structure of \( U \) by connecting some of its sprouts to some of its roots. Unfortunately, identity cyclic extensions do not suffice for that purpose, as identity-sharing extensions do for sharing, unless in special cases:

Example 28. Let \( L \) be the drag made of two copies of the tree \( f(x) \) sharing the variable \( x \). Three identity extensions of \( L \) are represented in Figure 5. The first is the trivial directed extension, while the two others are cyclic ones. The second maps the variable \( x \) of \( L \) to the only root of the identity drag, which is its sprout \( y \), and the only sprout \( y \) of the identity drag to the first root of \( L \). The third instead maps \( y \) to the second (or third) root.

Identity cyclic extensions allow one to change the structure of a drag by adding new edges without changing its internal nodes. If the drag has a single root, it is easy to see that identity extensions are enough to predict all forms that

\[ L = L \otimes_{\Theta_0} 1^Z_0 \]

\[ L \otimes_{\{x \mapsto 3, y \mapsto 1\}} 1^Z_0 \]

\[ L \otimes_{\{x \mapsto 3, y \mapsto 2\}} 1^Z_0 \]

Figure 5: Identity cyclic extensions.
a drag may take under composition with an extension. This is no longer true with multirooted drags, since identity cyclic extensions cannot reach two different roots from the same sprout.

Note that non-identity cyclic extensions cannot do any better if the vocabulary contains unary symbols only, as in Example 28. A cyclic extension must then have a single vertex as root, possibly repeated, since \( L \) has a single sprout, and all internal vertices of the extension must be accessible from that sprout. Then, due to the vocabulary, the extension will cover a single root of \( L \). If, however, the vocabulary contains a binary symbol \( g \), then new cyclic extensions pop up, both with the same drag (the extension will cover a single root of \( L \)) and all internal vertices of the extension must be accessible from that sprout. Then, due to the vocabulary, the extension will cover a single root of \( L \).

The following property is at the heart of drag rewriting:

**Lemma 29 (Decomposition).** Let \( U,W \) be clean nonempty drags and \( \xi \) be a switchboard for \( W,U \). If \( \xi_W \) is onto \( R^*(U) \), then, there exist drags \( A,B,C \) and switchboards \( \zeta,\theta \) such that

1. \( B \langle \xi_B,\xi_{U\rightarrow B} \rangle \) is a cyclic extension of \( U \) denoted by \( B\xi \);
2. \( C\theta \) is a substitution extension of \( B\otimes\xi U \);
3. \( A\zeta \) is a context extension of \( (B\otimes\xi U)\otimes\theta C \);
4. \( W\otimes\xi U = A\otimes\zeta ((B\otimes\xi U)\otimes\theta C) \);
5. \( C \) is empty if all internal nodes of \( W \) reach one of its sprouts;
6. \( B\otimes\xi U \) is cyclic if \( C \) is empty and \( \xi_B \) is onto \( R^*(U) \).

Note that \( A\zeta \) is a context extension of \( (U\otimes\zeta B)\otimes_{\xi} C \), not of \( U\otimes\zeta B \). Therefore, associativity of composition does not apply here.

**Proof.** The lemma is depicted in Figure 6. First, let \( D \) be the subdrag of \( W\otimes\xi U \) generated by \( R^*(U) \). Since \( U \) is clean, \( D \) must contain all vertices of \( U \) by definition of \( R^*(U) \). By Lemma 16, \( W\otimes\xi U = A\otimes\zeta D \) for some context extension \( A\zeta \) of \( D \) such that \( \zeta \) is total, surjective, and directed. Observe that \( \zeta_{A\rightarrow U} = \xi_A \). Now let \( C \) be the subdrag of \( D \) generated by those vertices that cannot reach a sprout in \( Dom(\xi_W) \), hence a root of \( U \). By definition of \( C \) and the fact that \( D \) contains all vertices of \( U \), the ancestor of \( C \) in \( D \) must contain all vertices of \( U \), hence is of the form \( B\otimes(\xi_{\theta}(\xi_{U\rightarrow a})) \). By Lemma 15, \( \theta = (B\otimes\xi_U)\otimes\theta C \) for some \( \theta \) that is total, surjective, and directed. Observe that \( \theta_{U} = \xi_{U\rightarrow C} \). Property 4 now follows. Properties 5 and 6 are straightforward. \( \square \)

This lemma applies of course to rewriting extensions \( W\xi \) of a drag \( U \) since, in this case, \( \xi_W \) must be onto all roots of \( U \).

A consequence of Lemma 29 is that properties of extensions are preserved by rewriting. Further, the decomposition lemma (Lemma 29) itself is also preserved, except when right-hand sides of rules lose variables from the left-hand side, in which case it needs to be slightly modified:

![Figure 6: Decomposition of a composition W ⊗_ξ U for which ξ_W is onto R*(U).](image-url)
Lemma 30. Let $U \rightarrow_{L} U'$. Then, $U = A \otimes_{\xi} ((B \otimes_{\xi} L) \otimes_{\theta} C)$ and $U' = A \otimes_{\xi} ((B \otimes_{\xi} R) \otimes_{\theta} C)$ for some $A, B, C$ and $\xi, \theta$, such that

1. $\theta$ is a substitution extension for both $B \otimes_{\xi} L$ and $B \otimes_{\xi} R$;
2. $A \otimes_{\xi}$ is a context extension for both $(B \otimes_{\xi} L) \otimes_{\theta} C$ and $(B \otimes_{\xi} R) \otimes_{\theta} C$; and
3. $B \otimes_{\xi}$ is a cyclic extension for $L$, and for $R$ as well if $\varphi(R) = \varphi(L)$. In case $\varphi(R) \subseteq \varphi(L)$, then $B = A' \otimes_{\theta} B'$ for some `drags' $A'$, $B'$ such that $B' \otimes_{\xi}$ is a cyclic extension of $R$ and $A' \otimes_{\theta}$ is a context extension of $B' \otimes_{\xi} R$.

Proof. By definition of rewriting, $U = W \otimes_{\xi} L$ and $U' = W \otimes_{\xi} R$, where $W \otimes_{\xi}$ is a rewriting extension of $L$. By the definition of a rewriting extension, Lemma 29 applies; hence, $W \otimes_{\xi} L = A \otimes_{\xi} (B \otimes_{\xi} L) \otimes_{\theta} C$. By the definition of rewriting, $B \otimes_{\xi} L \rightarrow B \otimes_{\xi} R$, hence $\varphi(V(B \otimes_{\xi} R)) \subseteq \varphi(V(B \otimes_{\xi} L))$ and $|\varphi(B \otimes_{\xi} R)| = |\varphi(B \otimes_{\xi} L)|$. By Lemma 25. It follows that $\theta$ is a switchboard for $C, B \otimes_{\xi} R$.

Claim follows. The proof of (2) is similar. We are left with (3).

7. Drag matching

Matching is the fundamental operation that is used for rewriting. In practice, matching is expected to be fast, at least on average. For terms, matching is linear, and constant time on average (most matching attempts fail quickly, without reading the whole input).

Given a `drag' $U$ to be reduced by a rewriting system $R$, we need to find a rule $L \rightarrow R \in R$ and a rewriting extension $W \otimes_{\xi}$ of $L$ such that $U = W \otimes_{\xi} L$. By Lemma 29, $U = A \otimes_{\xi} ((B \otimes_{\xi} L) \otimes_{\theta} C)$ for some extensions $B \otimes_{\xi}, C \otimes_{\theta}$, and $A \otimes_{\xi}$. This will allow us to control the search for $W \otimes_{\xi}$. In particular, this being directed, the head of $B \otimes_{\xi} L$ must be a head occurring in the dag decomposition of $U$, which can be easily obtained from $U$ in linear time by a depth-first search.

We now formalize this intuition:

**Lemma 31.** Given a rewrite rule $L \rightarrow R$, let $W \otimes_{\xi}$ be a rewriting extension of $L$ such that $U = W \otimes_{\xi} L$. Then, either $\nabla U = \nabla W \otimes_{\xi} L$ or there exist $U' \subseteq \overline{U}$ and a cyclic extension $B \otimes_{\xi}$ of $L$ such that $B \otimes_{\xi} L = U'$.

**Proof.** By Lemma 25, $U = A \otimes_{\xi} ((B \otimes_{\xi} L) \otimes_{\theta} C)$, where $\xi$ is directed. There are two cases:

1. If $A$ is empty, then $U = (B \otimes_{\xi} L) \otimes_{\theta} C$. Since $\theta$ is directed, then $B \otimes_{\xi} L \subseteq \overline{U}$.
2. Otherwise, $A = \overline{A} \otimes_{\theta} \nabla A$ where $\delta$ is directed. By Theorem 19, $\nabla U = \nabla A \otimes_{\xi} ((B \otimes_{\xi} L) \otimes_{\theta} C) = \nabla W \otimes_{\xi} L$, and we are done.

This lemma relates (a subdrag of) the head of $D$ with the head of $B \otimes_{\xi} L$, where $B$ is unknown and $\xi$ is the entire left-hand side. The consequence is that matching drags is similar to matching terms, the notion of matching a term at some position being replaced by matching a drag at some subdrag.

We now refine this lemma so as to relate the head of $D$ with the head of $B$.

**Lemma 32.** Given a cyclic extension $B \otimes_{\xi}$ of $L$, there exist two subdraggs $L_{\xi}$ and $B_{\xi}$ of $L$ and $B$, respectively, such that $(L \otimes_{\xi} B) = L_{\alpha} \otimes_{\xi} B_{\xi}$, where $L_{\alpha}$ and $B_{\xi}$ are the antecedents of $L_{\alpha}$ and $B_{\xi}$ with respect to $L$ and $B$, respectively, called the anticipated head and the anticipated tail of $L$ with respect to $B \otimes_{\xi}$, respectively.

**Proof.** By Lemma 19, $L \otimes_{\xi} B = (L \otimes_{\xi} B) \otimes_{\xi} \nabla (L \otimes_{\xi} B)$, where $\xi$ is a directed switchboard. The vertices of $\nabla (L \otimes_{\xi} B)$ define two subdraggs, one of $L$ and one of $B$, which we call $L_{\alpha}$ and $B_{\xi}$, respectively. Let $L_{\alpha}$ and $B_{\xi}$ be their respective antecedents in $L$ and $B$ defined in Lemma 16. Then, referring again by $\xi$ to the restrictions of the directed switchboard $\xi$ to $L_{\alpha}, B_{\xi}$, then $(L \otimes_{\xi} B) = L_{\alpha} \otimes_{\xi} B_{\xi}$. Note further that $B_{\xi} \otimes_{\xi}$ is a cyclic extension of $L_{\alpha}$.

We now turn to a last question before sketching the matching algorithm. Can we precompute all possible $L_{\alpha}$?

To this end, we need to know which part of $L$ is used to build the head of $B \otimes_{\xi} L$. We already addressed this question in Section 5 by computing identity cyclic extensions of $L$. We also pointed out that identity cyclic extensions do not suffice unless the vocabulary is too poor to build branching structures. As a remedy, we introduce a new symbol $\sqcup$ of a variable arity which is meant to abstract the internal vertices of a particular cyclic extension:
Definition 33. Given a drag $L$ whose list of roots has length $n$ and sprouts are labeled by $x_1, \ldots, x_m$, we call any cyclic extension $\square^n_0 \xi$ of $L$ an extensible (cyclic) extension, where

1. $\square^n_0$ is the drag which has a unique internal vertex labeled $\square$ (and also named $\square$), $q \leq n$ sprouts identified with the distinct variables $y_1, \ldots, y_q$ that label them, and a list of roots of length $p \leq m$;
2. $\xi$ is total map from $y_1, \ldots, y_q$ to $1 \ldots n$ and $\xi_L$ is a surjective map from $x_1, \ldots, x_m$ to $1 \ldots p$.

Since there are finitely many extensible cyclic extensions of a drag $L$, up to the renaming of its variable-sprouts, matching drags will be decidable. Note that the internal vertex $\square$ need not be a root, in which case cleaning the extensible extension gives an identity extension, so that extensible extensions truly generalize identity extensions. In particular, an empty list of roots given the trivial extensible extension, which is then the empty drag.

Given an extensible cyclic extension of $L$, we can compute its head and tail. By design, the vertex $\square$ of $L \otimes \xi$ must be part of the head of $L \otimes \xi$ or of its tail. We get the following property:

Lemma 34. Given a cyclic extension $B \xi$ of $L$, there exists a extensible extension $\square \xi$ of $L$ such that the anticipated heads of $L$ with respect to $B \xi$ and $\square \xi$ are isomorphic.

Proof. It suffices to replace $L$ by the appropriate extensible cyclic extension $\square \xi$ in the construction given in the proof of Lemma 32.

Example 35. Consider the first drag $L$ of Figure 7, obtained by a tiny modification of the drag of Figure 28. We compute here its compositions with two extensible extensions, instead of the trivial one. The first illustrates that identity cyclic extensions achieve the same effect when they can be used. The second illustrates the use of a hyper-extension that cannot be simulated by a cyclic identity extension.

The matching algorithm for drags is based on Lemmas 31, 32, and 34. Its description below is nondeterministic: it is indeed in NP. In practice, however, left-hand sides of rules are usually quite small, which makes the complexity issue of secondary importance.

Question. Does a drag $D$ rewrite via rule $L \rightarrow R$?

Algorithm.

- Step 1: Try matching $L$ at the head of $D$. If it fails, recur with the tail of $D$ (Lemma 31).
- Step 2: Guess nondeterministically an anticipated head $H$ of $L$ (Lemma 34).
- Step 3: Verify that $H$ is included in $\hat{D}$ (Lemma 32).
- Step 4: Recur with the anticipated tail of $L$ and the tail of $D$ (Lemma 32).

The only aspect of this algorithm that has not been described in detail is the inclusion of $H$ in $\hat{D}$ at Step 3. Standard algorithms for checking the isomorphism problem for rooted directed graphs can be easily modified for that purpose so that the resulting algorithm is again linear. The details are left to the implementor.
8. Related frameworks

Composition of terms, that is, substitution, can be viewed in two different ways. The first, traditional way is to view composition as a homomorphism. This uses the fact that terms are freely generated. In this view, variables denote terms. The second is to interpret variables as pointers. In this graphical view, composition amounts to redirecting pointers. This view fits very well indeed with dags, whose algebraic structure is not freely generated. It is commonly adopted by developers for efficiency reasons, and this is why term-rewriting packages are often based on dags rather than trees. It is also the essence of interaction nets, originating from linear logic, which were introduced by Lafont long ago [23].

Our notion of drag composition coincides with this latter viewpoint: roots are input ports, sprouts are output ports, and composition is a redirection of pointers from output ports to input ports. This process-algebra jargon reflects the fact that variables do not denote graphs, but channels connecting precisely two processes, in the process algebra tradition, rather than graph hyper-edges that would connect many processes all together.

As we have already said, this view fits very well with dags, a bit less so with trees. (See, however, the discussion in the conclusion.) We believe that the theory of term rewriting scales to drags very smoothly, as it does to dags, and we have demonstrated it with the framework first (here), and also with the design of syntactic rewrite orderings in [9]. A specific, important benefit of our framework is that rewriting drags cannot create dangling pointers.

The traditional view of composition of graphs initiated by Ehrig et al. [12] takes the first route. Even though graphs are not freely generated, graph homomorphisms lead the way, rules being defined as pairs of homomorphisms preserving some interface graph included in the left-hand side, rewriting being defined as a double graph pushout preserving that interface, one for the left-hand side homomorphism, and one for the right-hand side homomorphism. Called for that reason the double-pushout approach (DPA), it can be described either in categorical terms or by means of a graph-gluing construction. An easy-to-read recent survey is [22], in which the gluing construction is explained in all its (many) details.

In this view, the interface graph can be anything (provided it is included in the left-hand side of the rule), and as a result, rewriting can result in dangling pointers. Such rewrites may be either forbidden, or otherwise require the elimination of those edges. The rewriting definition comes therefore along with various side conditions that have to be checked when calculating the pushouts. This aspect resulted in several variations of the initial framework, some of which are thoroughly analyzed in [15]. In particular, there is a debate whether the right-hand side homomorphism should be injective or not. For implementation purposes, injectiveness is of course much more comfortable.

Because rewriting is based on graph homomorphisms, termination methods based on interpretations can be applied without too much difficulty, as is demonstrated, in particular, in [5]. Testing local confluence, on the other hand, is much more delicate: it is in general undecidable, and leads to the computation of possibly infinitely many critical pairs for restricted decidable cases [26,11,24,13].

In the traditional view of graph rewriting, variables should stand for graph homomorphisms. Since this does somewhat complicate the framework, and in particular pattern matching a graph against a left-hand side homomorphism, variables have been absent from most works on graph rewriting of which we are aware. Exceptions are [25,28,16]. In these two works, hypergraphs are considered instead of graphs, and rewriting is again defined via a double pushout of hypergraphs. Variables, however, stand for edges of hypergraphs, with a given incoming and outgoing arity. This looks a little bit like our switchboard device, with the difference that a switchboard is a collection of one-way channels, which allows us to have variables as labels of vertices without successors, as in trees or graphs, and to control easily how nodes must be merged (or glued, in DBA jargon).

We can indeed view drag rewriting as a specific, very simple instance of Ehrig’s framework: a drag rewrite rule is simply a pair of drag homomorphisms; this follows from Lemma[30] the interface graph being restricted to the list of root vertices and the set of sprout vertices of the left-hand side drag. This choice of interface, also advocated by Kahl in a more restricted setting [20], has ideal behavior, by ensuring the absence of dangling pointers, and also making the double pushout construction trivial. The only remaining difficulty is matching, unless it is still easier than in the general DPA framework.

In order to compare now the two frameworks at work, we compare the use of drags to specify a variant of the distributed leader election algorithm of Chang and Roberts [6]. For the DBA specification, we refer to [22], where it is given by means of diagrams, as is usually the case for DPA.
Example 36 (Leader Election). First, the ring itself and its management: There are two connected components, each one being a rooted drag whose root is the name of the data structure used for implementing the protocol: one for the ring of processes, and one for the counter. We will use the symbol self to indicate an edge going back to the root of the data structure (here, the ring). This allows us to write all rules as algebraic expressions, the root being convention associated to the outermost symbol of the expression.

The vocabulary uses four sorts (with the usual algebraic meaning): natural numbers generated by 0 and s for representing process IDs, bool (T and F) used as a flag, ring, and counter. Messages are process IDs, hence are also represented by natural numbers. Booleans and natural numbers are assumed. Function symbols obey the following sort declarations:

\[
\begin{align*}
\text{nil} & : \text{self} \rightarrow \text{ring} \\
\text{id} & : \text{nat} \times \text{bool} \times \text{nat} \times \text{ring} \rightarrow \text{ring} \\
\text{co} & : \text{nat} \rightarrow \text{counter}
\end{align*}
\]

Initially, there is just

\[
\text{co}(0) \quad \text{nil}(\text{self})
\]

The ring is empty, hence has the form nil(self), and the shared counter holds the initial value 0.

The arguments in id(x, b, m, z) have the following meaning: x is the process ID; b becomes true when x is elected (this is a private argument); m is a process ID circulating in the ring to be checked for minimality; z is (a pointer to) the rest of the ring.

Each process can communicate its ID to the next process in the ring, and messages can then be passed along the ring past processes whose ID is larger. Otherwise, the message may be deleted. When and if a message returns to its sender, she becomes the leader. Any process can quit the ring at any time, unless it has been elected leader.

Although we have not defined conditional rules, the following rules use conditions, which should be interpreted in the very same way as conditional term-rewriting rules normally are. (Or they may be viewed as defining an infinite schema of instances. Or the conditions could be eliminated at the price of adding additional rules to implement the comparison between natural numbers.)

\[
\begin{align*}
\text{enter} & : \text{co}(x) \quad \text{nil}(\text{self}) \quad \rightarrow \quad \text{co}(s(x)) \quad (s(x), F, 0, \text{self}) \\
\text{enter} & : \text{co}(y) \quad \text{id}(x, b, m, z) \quad \rightarrow \quad \text{co}(s(y)) \quad \text{id}(x, b, m, \text{id}(s(y), F, 0, 0, z)) \\
\text{quit} & : \quad \text{id}(x, F, m, z) \quad \rightarrow \quad z \\
\text{send} & : \quad \text{id}(x, F, m, z) \quad \rightarrow \quad \text{id}(x, F, x, z) \\
\text{pass} & : \quad \text{id}(x, F, m, \text{id}(y, F, n, z)) \quad \rightarrow \quad \text{id}(x, F, m, \text{id}(y, F, m, z)) \quad \text{if} \quad x \geq m \\
\text{erase} & : \quad \text{id}(x, F, m, z) \quad \rightarrow \quad \text{id}(x, F, 0, z) \quad \text{if} \quad x < m \\
\text{elect} & : \quad \text{id}(x, F, x, z) \quad \rightarrow \quad \text{id}(x, T, 0, z) \\
\text{ignore} & : \quad \text{id}(x, T, m, z) \quad \rightarrow \quad \text{id}(x, T, 0, z)
\end{align*}
\]

The rules are nonterminating since processes can enter the ring without restriction, even if a leader has been elected already. All processes can also quit, but not the leader if already chosen. As a result, the ring may become empty, and then grow again. Note that it would be very easy to modify the protocol so as to allow the leader to quit as well (replace F by a don’t care variable in the left-hand side of rule quit).

This example demonstrates that the drag model is quite easy to use, thanks to the presence of variables. It is quite expressive too, although this example uses only a little bit of its capabilities. The use of self allows us to write drag expressions as if they were trees. We expand on this idea in the next section.

9. Drag expressions

It is indeed easy to give a general notation for drags in the style of Ariola and Klop’s notation for λ-graphs [1] by allowing for multi-rooted expressions. The main idea is to view the vertices of a drag as variable names and an incoming edge to a vertex v labeled by f as an equation of the form v = f(u) when they originate from the list of vertices u. The whole drag becomes therefore a set of such equations, in which each internal vertex v appears exactly once on the left of an equation. Note that these equations correspond to Prolog’s substitutions with occurs-check. Then, a scoping operator allows us to give structure to the whole set of equations; more precisely, it allows one to
reveal the dag structure of the graph by moving the scoping operator down to the leaves of the graph as long as is allowed by the scoping rules.

There is a vast literature around this idea, where the binding operator is sometimes called $\mu$ because of its greatest-fixpoint semantics \[^{29,12}\], while the expressions themselves are sometimes called term graphs because they implicitly have a single root \[^{27}\]. Our binder has a different semantics, which justifies a different binder name.

These languages of graph expressions were used by Ariola and Klop \[^{1}\], who studied confluence problems in graph rewriting, and by Goubault \[^{14}\], who studied termination. All these languages, however, fail to account for horizontal sharing, because term-graphs have a single root. Our language of expressions is a generalization that includes multiple roots. A syntax for composing expressions whose operations have multiple outputs was proposed by James Kajiya \[^{21}\].

9.1. Expressions

**Definition 37** (Drag Expressions). The set of expressions, denoted $G(\Sigma, \Xi)$, is defined by the grammar:

$$s, t := x | f(\overline{x}) | [x = f(\overline{x})]t | \overline{t}$$

with $f \in \Sigma$, $x \in \Xi$, $|\overline{s}| = |f|$, $|\overline{t}|_r \neq 0$, and $\Xi$, a list of pairwise distinct variables in $|x = f(\overline{x})|$, where the root-length $|s|_r$ of a drag expression $s$ (the number of its roots) is defined by induction as follows:

$$|s|_r = |f(\overline{s})|_r \equiv 1, \quad |[x = f(\overline{s})]t|_r \equiv |t|_r, \quad \text{and} \quad |\overline{t}|_r \equiv |\Xi|_{ref} |t|_r$$

Expressions are categorized as follows:

- $f(\overline{s})$ is root-algebraic, whose terms $\overline{s}$ are either variables $x$ or root-algebraic expressions all of whose strict subexpressions are themselves terms;
- $[x = f(\overline{s})]t$ is an abstraction whose assignment $x = f(\overline{s})$ is a (possibly empty) set and body $t$, an arbitrary expression.

Abstractions are expressions formed with the binder “$[\ldots]_r$” of arity $n > 0$, where $n = |\overline{s}|$ is usually omitted. The use of parentheses for lists of expressions is necessary to obtain a nonambiguous grammar that can be parsed efficiently. Note that there is no syntax for the empty drag (which could have been the empty list of drags, that is (), were it allowed by the grammar rules).

The root-length $|\overline{t}|_r$ of a list $\overline{t}$ should not be confused with the length of $\overline{t}$ denoted by $|\overline{t}|$. They are in general different. By convention, we shall usually characterize a list of length 1 by writing (t) instead of $\overline{t}$.

The first two grammar rules suffice for trees. Abstractions are needed for representing sharing. In the case of dags, or term graphs, the third grammar rule should be restricted so as to avoid cycles, whose presence can be detected by computing the occurs-check relationship to be defined later. The first three grammar rules define uni-rooted drags, the last grammar rule being necessary to allow for multi-rooted drags. Note also that variables in assignments can only be equated to root-algebraic terms; this is why their root-length is also 1. This syntax restriction eases technical developments in several ways.

In the following, we sometimes identify $[S](t)$ with $[S]t$, $[\overline{t}](t)$ with $\overline{t}$, and $[t]$ with $t$. These identifications are part of the congruence on drag expressions to be introduced in Section 9.3

**Definition 38** (Free and Bound Variables). A variable $x$ occurs free in a term $t$ iff any of the following holds:

1. $t = x,$ or
2. $t = f(\overline{t})$ and $x$ occurs free in $\overline{t},$ or
3. $t = [\overline{s}]x$ and $x$ occurs free in some $v \in \overline{s},$ or
4. $t = [x = s(\overline{t})]u$, $x$ occurs free in some $v \in \overline{s}$ or in $t,$ and $x \not\in \Xi$.

Otherwise, it is bound. We denote by $\text{Var}(t)$ the set of all variables occurring in the term $t$ and by $\Sigma \text{Var}(t)$ the subset of those occurring free. The set of ground expressions with no free variables, also called drag expressions, is denoted $G(\Sigma)$. 

18
<table>
<thead>
<tr>
<th>Drag</th>
<th>Drag Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>( f(g(b), a, g(b)) )</td>
</tr>
<tr>
<td>2nd</td>
<td>([x = b] f(g(x), a, g(x)))</td>
</tr>
<tr>
<td>3rd</td>
<td>([x = g(b)] f(x, a, x))</td>
</tr>
<tr>
<td>4th</td>
<td>(f(a, [x = g(b)](x, x)))</td>
</tr>
<tr>
<td>5th</td>
<td>([x = b] f(g(x), [y = g(z), z = h(y, x)](y, z)))</td>
</tr>
<tr>
<td>6th</td>
<td>([x = g(x), y = h(y', z), z = h(y, z'), z' = g(y'), z'' = h(y', z'), x' = f(x, y, y][x'])</td>
</tr>
<tr>
<td>7th</td>
<td>([x = b] f(x, [y = h(x, z), z = h(y, z'', z = g(y)z'')](y, z)))</td>
</tr>
</tbody>
</table>

Table 1: Drag expressions for the drags shown in Figure 1

**Definition 39 (Roots).** The list of roots \( \mathcal{R}(t) \) of a clean drag expression \( t \) is defined by induction as follows:

1. \( \mathcal{R}(x) = x \); 
2. \( \mathcal{R}(f(t)) = f(\mathcal{R}(t)) \); 
3. \( \mathcal{R}([t]) = \mathcal{R}(t) \); 
4. \( \mathcal{R}(x = f([t])t) = \mathcal{R}(t) \).

It can be easily verified that the root-length of a drag expression \( t \) is just the length of its list of roots.

**Example 40.** The seven drags shown in Figure 1 have expressions (among many others) shown in Table 1.

It is easy to match a drag with its expression. Note first that bound variables are used to name vertices, in particular shared vertices for which they are necessary. The first drag is a tree; no abstraction is needed. The second and third specify the sharing in an abstraction coming first, which gives a quite readable expression, which we call flat and third specify the sharing in an abstraction coming first, which gives a quite readable expression, which we call particular shared vertices for which they are necessary. The first drag is a tree; no abstraction is needed. The second drag is built by induction on the expression \( \mathcal{R}(t) \), namely, that the roots of \( \mathcal{R}(t) \) share some sprouts. Then \( \mathcal{R}(t) \) is reduced to its rooted sprout \( x \).

### 9.2. Conversions

We address here two related questions: Are all expressions drag representations (soundness)? Can all drags be represented (completeness)?

First, we describe the construction of a drag \([d]\) from an expression \( d \) with possibly free variables, hence giving semantics to our language of drag expressions, and answering therefore the first question. This construction ensures the root invariant property, namely, that the roots of \([d]\) are one-to-one with the roots of \( d \).

With no loss of generality, we may assume that no variable is bound twice in \( d \), nor free and bound. The drag \([d]\) is built by induction on the expression \( d \).

- \( d \in \Xi \). Then the drag \([d]\) is reduced to its rooted sprout \( x \).
- \( d = (\theta) \). By induction hypothesis, we build the list of drags \( d_i = [t_i] \), the drag \( d_i \) having a list of roots \( r_i \). Note that if a variable \( x \) appears several times in the list \( t \), then only one drag \([x]\) is constructed, meaning that the \( d_i \)'s share some sprouts. Then \([d]\) is the drag made by the collection of all \( d_i \)'s, whose list of roots is therefore the concatenation of the lists \( r_i \) in the order of their indices.
• \( d = f(\overline{s}) \). Then the drag \([d]\) will have, for its unique root, an internal vertex with new name \( r \) and label \( f \). By induction hypothesis, we build the drag \((\overline{s})\) whose list of roots \( \overline{r} \) will have length \(|\overline{r}| = |d_r| = |f|\). Then \([d]\) is obtained by adding an edge from \( r \) to each of the roots of the drag \([s]\). Note that the constraint on the number of outgoing edges from \( r \) is satisfied, hence \([d]\) is indeed a drag.

• \( d = [x = f(\overline{s})]t \). This is the nontrivial case, for which it is important that the different occurrences of a free variable \( x \) in \( t \) are represented by the same vertex in \([r]\). By induction hypothesis, let \([r]\) be the drag obtained from \( t \). Let now \( x \) be an arbitrary free variable of \( t \) which is bound in \( d \). Then, there exists an assignment \( x = f(\overline{s}) \) in \([x = f(\overline{s})]\). By induction hypothesis, the drag \( d_s = f(\overline{s}) \) constructed from \( f(\overline{s}) \) has a unique root \( r_s \). The drag \( d \) is then obtained by redirecting all edges of \([r]\) incident to \( x \) to the root \( r_s \) of \( d_s \), and this, for all such variable \( x \). The root invariant property is then easily verified.

Before addressing the second question, we need to define some basic notions and vocabulary:

**Definition 41** (Occurs-Check). Given an assignment \( \overline{x \equiv s} \), we define the following:

1. The occurs-check binary relation \( \prec_{oc} \) is the least transitive relation between variables in \( \overline{x} \) such that \( y \prec_{oc} z \) if \( z = v \in \overline{x \equiv s} \) and \( y \in \Sigma \mathit{Var}(v) \).
2. The set \( \cup y(\overline{x \equiv s}) \) of occurs-check variables of \( y \in \overline{x} \) is the set of variables \( \{ z : z \prec_{oc} y \prec_{oc} z \} \). We also use the short form \( \cup y \), assuming \( \overline{x} = \overline{s} \).
3. The set \( \cup (\overline{x \equiv s}) \) of occurs-check variables of \( \overline{x \equiv s} \) is the set \( \{ y : \cup y \neq \emptyset \} \).
4. A variable \( y \in \overline{x} \) is maximal if \( x \prec_{oc} y \Rightarrow y \prec_{oc} x \) for all \( x \).

Note that the meaning of the occurs-check relationship \( y \prec_{oc} z \) (as is the case in logic programming) is that the vertex \( y \) is further away from a maximal root than the vertex \( z \), hence the drag expression associated with the vertex \( y \) is smaller in size than that associated with the vertex \( z \). Our definition is made simpler than usual thanks to the assumption that terms in \( \overline{x} \) are root-algebraic, hence are not variables. Note also that a variable \( x \) that occurs only once in an assignment \( \overline{x \equiv s} \) has an empty set of occurs-check variables: the occurs-check relationship may not be reflexive. The reflexive closure of \( \cup \) is denoted \( \circ \).

**Definition 42** (Flat Assignment). An assignment \( \overline{x \equiv s} \) is flat if all expressions in \( \overline{x} \) are flat, that is, are either variables or are of the form \( f(\overline{y}) \) with \( \overline{y} \subseteq \overline{x} \). A drag expression in which all assignments are flat is said to be flat. A flat expression is said to be flattened if it is of the form \([S]x\) or \([S](\overline{t})\) where \( \overline{t} \in \Sigma \).

**Definition 43** (Clean Expression). A flattened expression \([S](\overline{x})\) is clean if \( \forall y \in \mathit{Var}(S) \) there exists \( \exists x \in \overline{x} \) \((y = x \lor y \prec_{oc} x)\).

Note that \( \overline{x} \) is therefore the set of maximal roots of a flattened expression \([S](\overline{x})\).

Constructing a flattened drag expression from a drag is easy: it suffices to name each vertex of the drag by a variable (new each time), then write down all equations that describe the relationship between a vertex and its outgoing neighbors, and finally build the expression as an abstraction whose roots can be easily listed since all vertices of the drag have a name. This gives a description of the drag that is flattened. Further, it is easy to see that this drag expression is clean if and only if the starting drag is.

Therefore:

**Theorem 44** (Adequacy). There is a correspondence from drag expressions to drags which is one-to-one from clean flattened drag expressions to clean drags.

9.3. Canonical expressions

We address a further complementary question in this section: The language of drag expressions being redundant, can we exhibit an equational theory that characterizes drag isomorphism? The main tool used for answering this question is the definition of normal forms by the means of convergent rewriting systems [8].

Whereas there is a single way to associate the drawing of a drag to a given drag expression by the previous construction, there are many ways to associate a drag expression to a given drag. The easiest method is the one already mentioned, which associates a variable to each vertex of the drag, writes all equations of the form \( x = f(\overline{y}) \)
expressing that the vertex \( x \) labeled by \( f \) has the vertices \( \vec{y} \) as successors, and outputs the expression \([x = f(\vec{y})](\vec{z})\)
where \( \vec{z} \) is the list (with repetitions) of variables denoting root vertices. But our language of expressions being richer than needed, there are many other ways: as we have seen with the examples, a drag may be denoted by many (actually infinitely many) different expressions. We therefore introduce an equational theory on drag expressions aimed at capturing drag isomorphism.

**Definition 45 (Convertible Expressions).** Two drag expressions \( s, t \) are convertible, written \( s \equiv t \), iff they are obtained from each other by using the following three sets of equations:

<table>
<thead>
<tr>
<th>( x = u )</th>
<th>( y = v )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t )</td>
<td>( t )</td>
</tr>
<tr>
<td>( { x = u, y = v } )</td>
<td>( { x = u, y = v } )</td>
</tr>
<tr>
<td>( { x = u, y = v } t )</td>
<td>( { x = u, y = v } t )</td>
</tr>
<tr>
<td>( { x = u, y = v } t )</td>
<td>( { x = u, y = v } t )</td>
</tr>
</tbody>
</table>

To decide convertibility, we are now going to transform this set of equations into a rewrite system that defines normal forms. It turns out that there are at least two ways to turn these equations into rewrite rules. In both cases, the first two equations, made of \( \alpha \)-conversion and permutation of elementary assignments, define the equational (modulo) part \( E \). The next four equations are oriented from left to right, yielding the set \( S \) of simplifiers, which serves for eliminating useless information, hence defining the clean normal forms. There are two choices for the last four equations: the factorization system \( FA \) oriented them from left to right, while the flattening system \( FL \) orients them from right to left, and adds rules that are needed to obtain the unique normal property:

\[
\begin{align*}
(\vec{v}, [y = s] t, w) & \rightarrow [y = s](\vec{v}, t, w) & \wedge( \alpha ) \\
\end{align*}
\]

Rules in \( FA \) and \( FL \) are supposed to apply to clean drag expressions, that is, to expressions in normal form with respect to the system \((S, E)\). Such a rewriting system \((R, S, E)\) coming into three parts, a set of equations \( E \), a set of simplifiers \( S \), and a set of rules \( R \), such that the rules apply to fully simplified expressions, is called normal [19].

**Theorem 46.** Both \((FA, NA, E)\) and \((FL, NA, E)\) are convergent (terminating and confluent) normal rewriting systems. Hence, they define the factorized and flat normal forms, respectively.

**Proof.** Checking convergence of normal rewriting systems is described in [19]. First, we need to prove that \( NA/E \) is a convergent rewriting system modulo. Termination is clear: \( E \) leaves the size of expressions invariant, while \( NA \) decreases it strictly. Further, their critical pairs (modulo \( E \)) are all joinable (modulo \( E \)). Confluence follows [18].

Next, we need to show that rewriting with the relation generated by \((FA, NA, E)\) on \( NA/E \)-normal forms is convergent. Termination of \( FA \) modulo \( E \) can be proved with a simple interpretation, using the size of expressions first for \((\alpha)\), and then a multiset of pairs \((m, n)\) interpreting the abstractions, \( n \) being the number of assignments in the abstraction, and \( m \) being the nesting level. The order on interpretations is therefore \( (>\alpha, (>\alpha, >\alpha))_{lex} \), which is invariant under \( E \). Again, all critical pairs are easily joinable.

Finally, we need to show that the relation generated by \((FL, NA, E)\) is convergent. Termination uses a similar interpretation, but the nesting level \( m \) is now subtracted from the maximum possible nesting level of a given expression. Again, critical pairs are easily shown joinable, once the two extra rules have been added.
We illustrate below the computation of the flat normal-form expression, by rewriting modulo the equations, of the expression given earlier for the seventh drag described in Table 1:

\[ x = b, y = h(x, z), z' = g(y) \]

\[ \rightarrow \Sigma \]

\[ [x = b] f(x, [y = h(x, z), z = h(y, z')][y, z]) \]

\[ \rightarrow \Sigma \]

\[ [x = b] f(x, [y = h(x, z), z = g(y), z = h(y, z')][y, z]) \]

\[ \rightarrow \Sigma \]

\[ [x = b, y = h(x, z), z' = g(y), z = h(y, z')][x, y, z'] \]

We now compute the factorized expression from the obtained flat expression, and observe that there are more steps in this direction:

\[ [x = b, y = h(x, z), z' = g(y), z = h(y, z')][x, y, z'] \]

\[ \rightarrow \Sigma \]

\[ [x = b] f(x, [y = h(x, z), z = g(y), z = h(y, z')][y, z]) \]

\[ \rightarrow \Sigma \]

\[ [x = b] f(x, [y = h(x, z), z = [z' = g(y)] h(y, z')][y, z]) \]

\[ \rightarrow \Sigma \]

\[ [x = b] f(x, [y = h(x, z), z = [h(y, z')][y, z]) \]

We can also observe that factorized normal forms are quite economical: all bound variables denote vertices of the graph that all have several incoming edges, hence are shared and must therefore be named. Further, every variable assignment in an abstraction is located as close as possible to its intended body, that is, at the vertex that is the closest common ancestor of the vertices denoted by its bound variables.

Because the equations of \( E \) need to be built into the matching process, rewriting with these systems could be expensive. This is not the case, though. Firstly, \( \alpha \)-conversion can be efficiently implemented. Secondly, left-hand sides of rules of \( S \) are linear, and matching modulo permutation can be done efficiently in that case. Thirdly, all rules in FL are left-linear, hence the same remark applies. For \( FA \), there is a single rule whose left-hand side is nonlinear, \( \rightarrow \). For that rule, however, it suffices to match first the variable \( y \) in root position, and then the rest of the left-hand side, which makes it even easier. The normal-form computations can therefore be efficiently implemented.

These normal forms can be used in many ways. First, as a theoretical tool, to show the main property of convertibility.

**Lemma 47** (Soundness). Convertible drag expressions denote isomorphic drags.

**Proof.** To this end, it is enough to show that this is true of all equalities that define convertibility. This uses the construction of a drag from an arbitrary drag expression given earlier. \( \square \)

Convertibility is a sound and complete axiomatization of drag isomorphism:

**Theorem 48.** Two drags are isomorphic iff their drag expressions are convertible.

**Proof.** Two drags are isomorphic iff there is a one-to-one map between their sets of vertices that identifies their respective lists of roots and commutes with their successor functions. We can therefore assume without loss of generality that their sets of vertices are identical subsets of \( \Xi \). Given two isomorphic drags whose vertex names are now identified, the corresponding drag expressions are both of the form \( \Xi = f(\Xi)[\Xi] \), hence can only differ in the name of their variables and permutation of their variable assignments. They are therefore convertible. Conversely, two convertible drag expressions have the same flat normal form (up to renaming of their variables and permutation of their assignments) by Theorem 46. The drag construction yields then the same drag for both. \( \square \)

The normal forms we have defined have many other uses. In particular, it is easy to characterize the following from the flat normal form \( d \downarrow = [x = f(y)][z] \) of a drag \( d \); the connected components of \( d \), the head of \( d \), the tail of \( d \). This is what we explain first. In all cases, we indeed obtain a flat drag expression from \( d \downarrow \) as a result of the construction.

For the connected components, we simply need to calculate for each variable:

1. The partition of \( \Xi \) with respect to the occurs-check equivalence.
2. For each obtained class, the restriction of \([x = f(\overline{x})]\) to the variables which are smaller than that of the class under the reflexive closure of the occurs-check relationship.

The complexity of the algorithm is that of the first phase, which is almost linear.

For the head \(d\), we simply need to restrict \([x = f(\overline{y})]\) to the variables which are minimal in the occurs-check relationship, yielding \(\hat{d} = [x' = f'(\overline{y}')](\overline{z}')\). For the tail, we need to restrict \([x = f(\overline{y})]\) to the variables which are not in \(\overline{z}'\) and add the proper roots, yielding \([x'' = f''(\overline{y}'')](\overline{z}'')\), where \(\overline{z}''\) is obtained by adding the variables of \(\text{Var}(x' = f'(\overline{y}'))\) \(\setminus\overline{z}'\) to those of \(\overline{z} \setminus \overline{z}'\) at the proper places (as specified by the definition of subdrag).

It turns out that the same computations are even simpler with the factorized normal form of \(d\), denoted by \(d\) itself. The head is then defined by induction over \(d\), using an environment initialized as the empty list of assignments. If \(d\) is a variable \(x\), its head is \(x\) if there is no assignment for \(x\) in the environment, and the head of \(s\) in the same environment if there is an assignment \(x = s\) in the environment. If \(d = f(\overline{s})\), its head is \(f(\overline{s})\), where \(\overline{s}\) is a vector of fresh variables. If \(d\) is the list \((\overline{s})\), its head is the concatenation of the heads of the \(s_i\)'s. And if \(d = [x = f(\overline{s})]\), then the head of \(d\) is the head of \(t\) in the environment augmented with the assignments \(x = f(\overline{s})\).

The computation of the tail of \(d\) can be defined by induction as well.

10. Conclusion

We have invented drags, a very general class of multigraphs, and explored their nice and useful properties. This has resulted in a new, simple, and very effective model for graph rewriting.

The drag model generalizes the term-graph model by allowing for cycles without any sort of restriction. But one can ask whether it can capture term rewriting per se? There is at least one simple way in which that can be achieved. Looking at rewriting as a relation between drags, the fact that variables that have multiple occurrences in drag rewrite rules are implemented with a copying or sharing operation is entirely determined by the definition of the switchboard:

\[
\begin{align*}
\text{equality in conditions } (1, 2) \text{ of Definition 6 yields sharing. Using isomorphism allows for copying. We could therefore index variables in right-hand sides of rules, so that variables having different indexes could not be shared; on the other hand, variables having the same index could possibly be shared. The absence of index would conventionally be index 0. Variables in left-hand sides would not be indexed, allowing them to be shared if necessary. Details remain to be worked out.}
\end{align*}
\]

In [9], we use this model to show that termination techniques that have been developed for terms, specifically, the recursive path order, scale to drags. The obtained order is generated from an order on heads (which are cyclic drags), in the same way as the recursive path order is generated from an order on function symbols. In turn, the order on heads is generated from an order on their function symbols. This order must enjoy a specific monotonicity property, when heads grow by adding new vertices. This work shows that the difficulty with rewrite orderings for graphs lies in monotonicity with respect to cyclic extensions: breaking, forming or growing cycles is indeed a new phenomenon that shows up with our drag model.

In the same paper, we have also succeeded in designing a total order for extending the theory of Gröbner bases to algebraic operads, which are polynomial expressions built over drags (making cyclic monotonicity vanish in this case). The rewrite order, on the other hand, is not total, and, we believe it cannot be.

We have not started to investigate confluence in our drag model. For terminating term-rewriting systems, confluence reduces to the joinability of critical pairs. This is true too for dag rewrite rules [26], that is, terms with sharing but without cycles.

The presence of cycles raises two difficulties: first, unification, but that should be the easy part. The second difficulty is the reduction of local peaks (pairs of rewrites of the form \(u \rightarrow v\) and \(u \rightarrow w\) to critical pairs obtained by overlapping left-hand sides of rules. In the tree case, this reduction depends heavily on the tree structure: two left-hand sides in a term are either: (i) at parallel positions, (ii) at related positions without overlapping, and (iii) at related positions with overlapping. With drags, these situations are no longer exclusive. We believe that the key to solving this problem lies in using the tree decomposition of a drag and Lemma [29].

A pleasant extension of our framework would be to allow for symbols of varyadic arity in order to facilitate rewriting modulo associativity and commutativity. This would be of interest, for example, for algebraic operads. We have not worked this out yet.

Our framework has the potential to be further extended with abstraction and application, in order to obtain a language for \(\lambda\)-terms with sharing and back-arrows, also called lambda graphs [1]. So far, we have designed a
language for drags inspired from this latter work by introducing the possibility of having many roots. Although introducing also application and abstraction is now easy, generalizing our drag model so as to obtain a meaningful λ-calculus for drags may conceal some unforeseen difficulties.

Acknowledgements

We thank Alfons Geser, who suggested the example that disclosed ill-behaved switchboards.

References


